NYU Abu Dhabi **Division of Engineering**

ENGR-UH 3511

Computer Organization and Architecture

Name:	Nishant Aswani
Net ID:	nsa325
Assignment Title:	Lab 3

Microprocessor Design and Verilog HDL: Part 1

Nishant Aswani, nsa325@nyu.edu

Computer Organization and Architecture(ENGR-UH 3511), Instructor Cristoforos Vasilatos

1 Introduction

The MIPS architecture falls under the reduced instruction set computer (RISC) family of instruction set architectures (ISAs). MIPS is a 32-bit architecture, employing 32 registers, each 32 bits wide. Verilog is a hardware description language, used to model hardware systems. Unlike C/C++, Verilog operates on modules and processes, the latter of which run in parallel, instead sequentially.

The following lab uses Verilog to begin an implementation of a 32-bit MIPS CPU. The modules implemented include a program counter, instruction memory, register file, ALU, and data memory.

2 Methodology

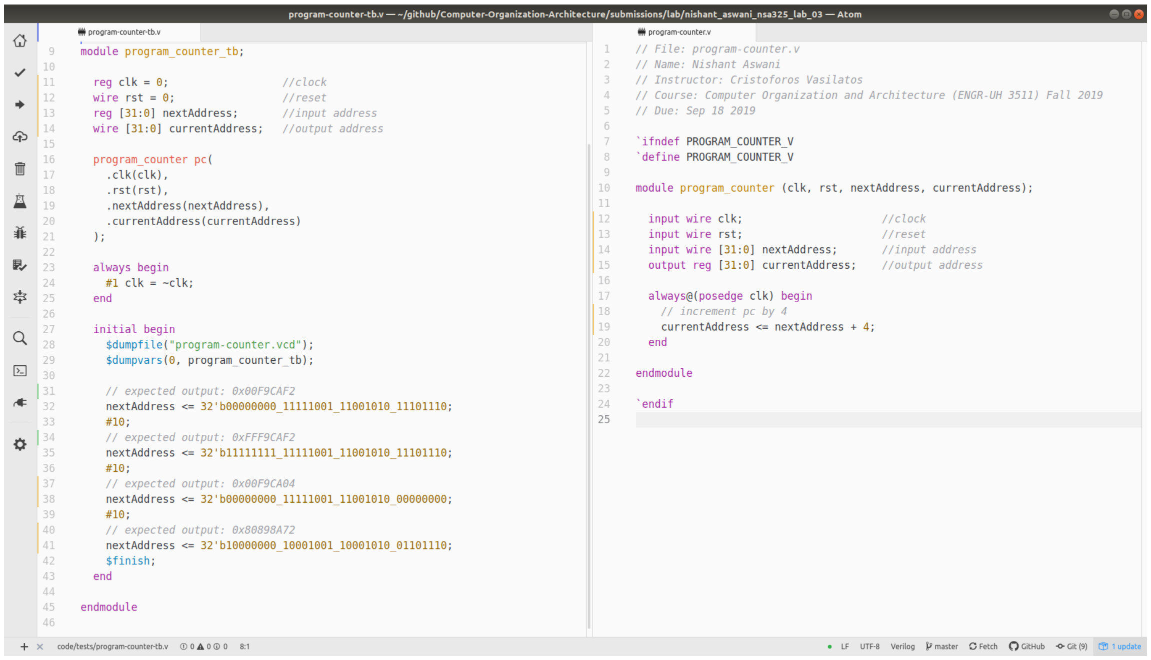
Icarus Verilog was used as a compiler to produce the object files for all verilog modules. The object file was then simulated using the `vvp` command from Icarus Verilog. The final output was viewed in GTKwave.

A make file was written to compile all .v files in the testbench folder. The makefile then runs `vvp` and `gtkwave`.

3 Results

3.1 Program Counter

Below is the implementation of the program counter, with the test bench on the left. As seen in the `always@` block, the logic simply adds 4 to the value of the input address to give the next instruction address. Hence, there is a clock input, an input address, and an output address.



```
module program_counter_tb;

  reg clk = 0;                //clock
  wire rst = 0;               //reset
  reg [31:0] nextAddress;     //input address
  wire [31:0] currentAddress; //output address

  program_counter pc(
    .clk(clk),
    .rst(rst),
    .nextAddress(nextAddress),
    .currentAddress(currentAddress)
  );

  always begin
    #1 clk = ~clk;
  end

  initial begin
    $dumpfile("program-counter.vcd");
    $dumpvars(0, program_counter_tb);

    // expected output: 0x00F9CAF2
    nextAddress <= 32'b00000000_11111001_11001010_11101110;
    #10;
    // expected output: 0xFFF9CAF2
    nextAddress <= 32'b11111111_11111001_11001010_11101110;
    #10;
    // expected output: 0x00F9CA04
    nextAddress <= 32'b00000000_11111001_11001010_00000000;
    #10;
    // expected output: 0x80898A72
    nextAddress <= 32'b10000000_10001001_10001010_01101110;
    $finish;
  end

endmodule
```

```
// File: program-counter.v
// Name: Nishant Aswani
// Instructor: Cristoforos Vasilatos
// Course: Computer Organization and Architecture (ENGR-UH 3511) Fall 2019
// Due: Sep 18 2019

`ifndef PROGRAM_COUNTER_V
`define PROGRAM_COUNTER_V

module program_counter (clk, rst, nextAddress, currentAddress);

  input wire clk;                      //clock
  input wire rst;                      //reset
  input wire [31:0] nextAddress;       //input address
  output reg [31:0] currentAddress;    //output address

  always@(posedge clk) begin
    // increment pc by 4
    currentAddress <= nextAddress + 4;
  end

endmodule

`endif
```

Figure 1: Verilog implementation of program counter with its testbench

The test bench simply defines a clock that alternates every unit of time. Next, it sends four different address, each time expecting an output that is 4 greater than the input.

The variable nextAddress refers to the input address, while the variable currentAddress refers to the output. As seen below, an input value, 0x00F9CAEE, is fed using "nextAddress". Adding four to the input value results in 0x00F9CAF2, which is displayed by the "currentAddress" module. Similarly, the program counter outputs input+4 at the positive edge of the upcoming clock cycle for each of the inputs.

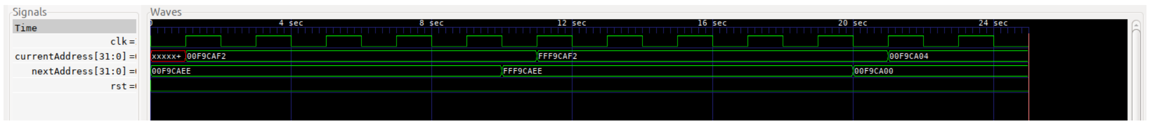
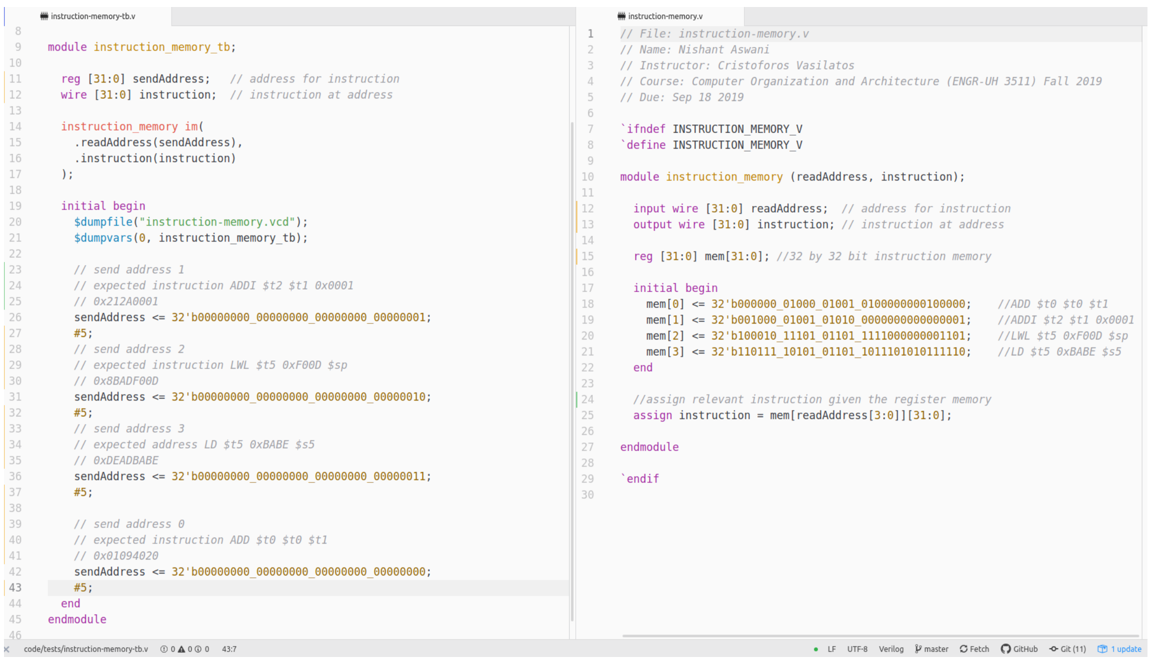


Figure 2: Waveform for program counter

3.2 Instruction Memory

To simulate memory, we use an array of 32 bit registers and instantiate a few of them with dummy instructions for demonstration purposes. Otherwise, the module simply takes in an address value, which is fed into the array as an index. It then retrieves the stored instruction.

The testbench asks for the instructions in registers 1, 2, 3, and 0 respectively. For example, registers 2 and 3 store hex values `0x8BADFOOD` and `0xDEADBABE`, which can be seen in the middle in the `gtkwave` screenshow below. Since the instruction memory is not driven by the positive edge of the clock in this implementation, it is able to retrieve the instruction without a clock cycle delay.


instruction-memory-tb.v

```verilog

module instruction_memory_tb;

  reg [31:0] sendAddress;   // address for instruction
  wire [31:0] instruction;  // instruction at address

  instruction_memory im(
    .readAddress(sendAddress),
    .instruction(instruction)
  );

  initial begin
    $dumpfile("instruction-memory.vcd");
    $dumpvars(0, instruction_memory_tb);

    // send address 1
    // expected instruction ADDI $t2 $t1 0x0001
    // 0x212A0001
    sendAddress <= 32'b00000000_00000000_00000000_00000001;
    #5;
    // send address 2
    // expected instruction LWL $t5 0xF00D $sp
    // 0x8BADF00D
    sendAddress <= 32'b00000000_00000000_00000000_00000010;
    #5;
    // send address 3
    // expected address LD $t5 0xBABE $s5
    // 0xDEADBABE
    sendAddress <= 32'b00000000_00000000_00000000_00000011;
    #5;

    // send address 0
    // expected instruction ADD $t0 $t0 $t1
    // 0x01094020
    sendAddress <= 32'b00000000_00000000_00000000_00000000;
    #5;
  end
endmodule

```

instruction-memory.v

```verilog
// File: instruction-memory.v
// Name: Nishant Aswani
// Instructor: Cristoforos Vasilatos
// Course: Computer Organization and Architecture (ENGR-UH 3511) Fall 2019
// Due: Sep 18 2019

`ifndef INSTRUCTION_MEMORY_V
`define INSTRUCTION_MEMORY_V

module instruction_memory (readAddress, instruction);

  input wire [31:0] readAddress;  // address for instruction
  output wire [31:0] instruction; // instruction at address

  reg [31:0] mem[31:0]; //32 by 32 bit instruction memory

  initial begin
    mem[0] <= 32'b000000_01000_01001_0100000000100000;    //ADD $t0 $t0 $t1
    mem[1] <= 32'b001000_01001_01010_0000000000000001;    //ADDI $t2 $t1 0x0001
    mem[2] <= 32'b100010_11101_01101_1111000000001101;    //LWL $t5 0xF00D $sp
    mem[3] <= 32'b110111_10101_01101_1011101010111110;    //LD $t5 0xBABE $s5
  end

  //assign relevant instruction given the register memory
  assign instruction = mem[readAddress[3:0]][31:0];

endmodule

`endif

```

code/tests/instruction-memory-tb.v 0 0 0 43:7 LF UTF-8 Verilog master Fetch GitHub Git (11) 1 update

Figure 3: Verilog implementation of instruction memory with its testbench

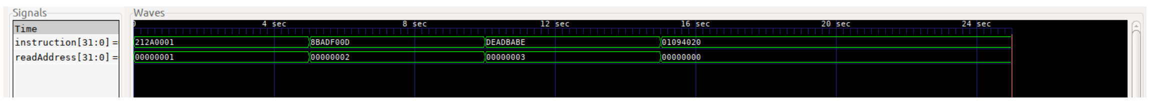
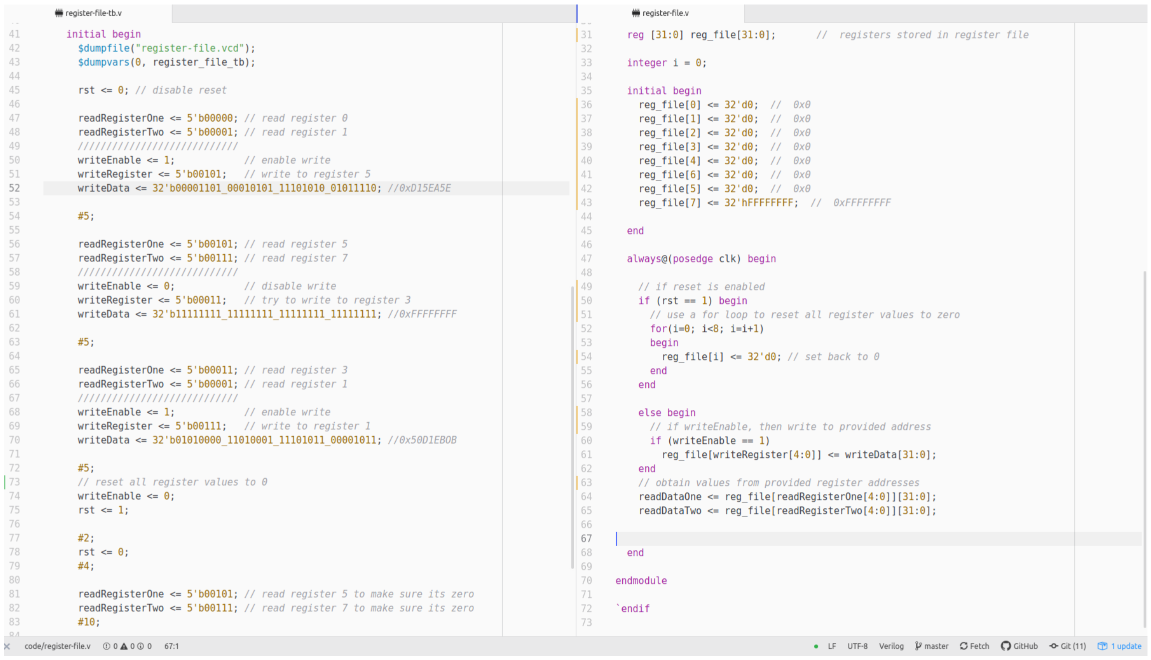


Figure 4: Waveform for instruction memory

3.3 Register File

The register file is the part of the CPU which stores 32 registers, each 32 bits wide. The module accepts 5 bit addresses for three registers (two for reading and one for writing). It also accepts 32 bits for the data to be written into the specified register. Moreover, there is a write enable control signal. Finally, the register file has two outputs, each 32 bits, which carry the value stored in the read addresses provided. Traditionally, these 32 bit values are sent off to the ALU for processing.

The module instantiates a 2D array for registers. Then, at each positive edge of the clock, it first checks for the reset signal. If reset is enabled, all registers are set to 0. Otherwise, it checks for the writeEnable to decide for or against storing the provided write data. At the end of this if/else block, the readData wires are assigned the corresponding values from the register file.


register-file-tb.v

```verilog
  initial begin
    $dumpfile("register-file.vcd");
    $dumpvars(0, register_file_tb);

    rst <= 0; // disable reset

    readRegisterOne <= 5'b00000; // read register 0
    readRegisterTwo <= 5'b00001; // read register 1
    ////////////////////////////
    writeEnable <= 1;            // enable write
    writeRegister <= 5'b00101;   // write to register 5
    writeData <= 32'b00001101_00010101_11101010_01011110; //0xD15EA5E

    #5;

    readRegisterOne <= 5'b00101; // read register 5
    readRegisterTwo <= 5'b00111; // read register 7
    ////////////////////////////
    writeEnable <= 0;            // disable write
    writeRegister <= 5'b00011;   // try to write to register 3
    writeData <= 32'b11111111_11111111_11111111_11111111; //0xFFFFFFFF

    #5;

    readRegisterOne <= 5'b00011; // read register 3
    readRegisterTwo <= 5'b00001; // read register 1
    ////////////////////////////
    writeEnable <= 1;            // enable write
    writeRegister <= 5'b00111;   // write to register 1
    writeData <= 32'b01010000_11010001_11101011_00001011; //0x50D1EB0B

    #5;
    // reset all register values to 0
    writeEnable <= 0;
    rst <= 1;

    #2;
    rst <= 0;
    #4;

    readRegisterOne <= 5'b00101; // read register 5 to make sure its zero
    readRegisterTwo <= 5'b00111; // read register 7 to make sure its zero
    #10;
```

register-file.v

```verilog
  reg [31:0] reg_file[31:0];       //  registers stored in register file

  integer i = 0;

  initial begin
    reg_file[0] <= 32'd0;  //  0x0
    reg_file[1] <= 32'd0;  //  0x0
    reg_file[2] <= 32'd0;  //  0x0
    reg_file[3] <= 32'd0;  //  0x0
    reg_file[4] <= 32'd0;  //  0x0
    reg_file[6] <= 32'd0;  //  0x0
    reg_file[5] <= 32'd0;  //  0x0
    reg_file[7] <= 32'hFFFFFFFF;  //  0xFFFFFFFF

  end

  always@(posedge clk) begin

    // if reset is enabled
    if (rst == 1) begin
      // use a for loop to reset all register values to zero
      for(i=0; i<8; i=i+1)
      begin
        reg_file[i] <= 32'd0; // set back to 0
      end
    end

    else begin
      // if writeEnable, then write to provided address
      if (writeEnable == 1)
        reg_file[writeRegister[4:0]] <= writeData[31:0];
    end
    // obtain values from provided register addresses
    readDataOne <= reg_file[readRegisterOne[4:0]][31:0];
    readDataTwo <= reg_file[readRegisterTwo[4:0]][31:0];


  end

endmodule

`endif
```

Figure 5: Verilog implementation of register file with its testbench

Notice in the implementation that all registers are instantiated with value 0, except register 7 which holds 0xFFFFFFFF. This is for demonstration purposes. Looking at the output, in the first cycle, we write the value `0xD15EA5E` to register 5. In the next cycle, we read register 5 and 7. Accordingly, the readDataOne and readDataTwo output `0xD15EA5E` and 0xFFFFFFFF, respectively.

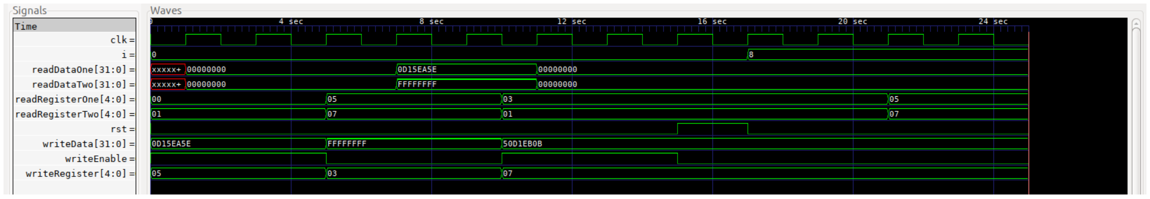


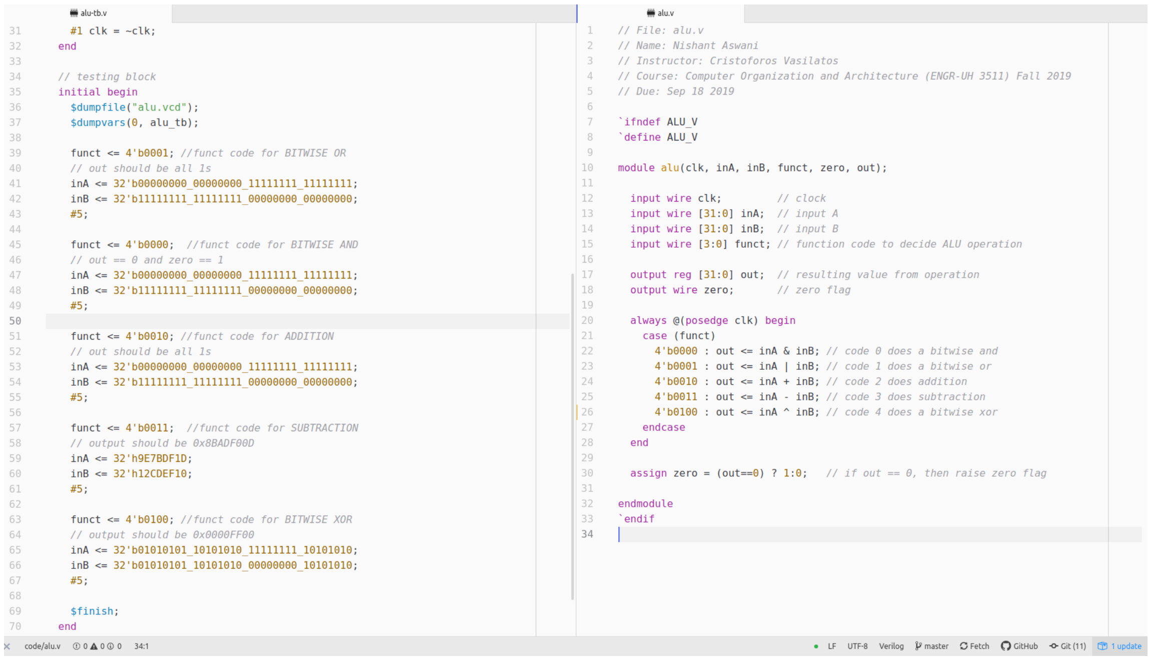
Figure 6: Waveform for the register file module

Within the same cycle, the code attempts to write to register 3. However, we see in the next cycle that register 3 remains unchanged, because the writeEnable signal was not turned on. Finally, the last cycle reflects the effects of enabling the reset buttons: register 5 and 7 have been reset to 0.

3.4 ALU

The ALU is driven by a clock and carries out an operation, specified by the function code, on two 32-bit values. It then outputs the resulting 32-bit value along with a flag which conveys whether the result is 0 or not.

As a result, the ALU implementation is a simple switch case, carrying out the operation corresponding to the function code. Finally, it carries out a simple if/else check for the zero flag.



alu-tb.v

```verilog
    #1 clk = ~clk;
  end

  // testing block
  initial begin
    $dumpfile("alu.vcd");
    $dumpvars(0, alu_tb);

    funct <= 4'b0001; //funct code for BITWISE OR
    // out should be all 1s
    inA <= 32'b00000000_00000000_11111111_11111111;
    inB <= 32'b11111111_11111111_00000000_00000000;
    #5;

    funct <= 4'b0000;  //funct code for BITWISE AND
    // out == 0 and zero == 1
    inA <= 32'b00000000_00000000_11111111_11111111;
    inB <= 32'b11111111_11111111_00000000_00000000;
    #5;

    funct <= 4'b0010; //funct code for ADDITION
    // out should be all 1s
    inA <= 32'b00000000_00000000_11111111_11111111;
    inB <= 32'b11111111_11111111_00000000_00000000;
    #5;

    funct <= 4'b0011;  //funct code for SUBTRACTION
    // output should be 0x8BADF00D
    inA <= 32'h9E7BDF1D;
    inB <= 32'h12CDEF10;
    #5;

    funct <= 4'b0100; //funct code for BITWISE XOR
    // output should be 0x0000FF00
    inA <= 32'b01010101_10101010_11111111_10101010;
    inB <= 32'b01010101_10101010_00000000_10101010;
    #5;

    $finish;
  end
```

alu.v

```verilog
// File: alu.v
// Name: Nishant Aswani
// Instructor: Cristoforos Vasilatos
// Course: Computer Organization and Architecture (ENGR-UH 3511) Fall 2019
// Due: Sep 18 2019

`ifndef ALU_V
`define ALU_V

module alu(clk, inA, inB, funct, zero, out);

  input wire clk;         // clock
  input wire [31:0] inA;  // input A
  input wire [31:0] inB;  // input B
  input wire [3:0] funct; // function code to decide ALU operation

  output reg [31:0] out;  // resulting value from operation
  output wire zero;       // zero flag

  always @(posedge clk) begin
    case (funct)
      4'b0000 : out <= inA & inB; // code 0 does a bitwise and
      4'b0001 : out <= inA | inB; // code 1 does a bitwise or
      4'b0010 : out <= inA + inB; // code 2 does addition
      4'b0011 : out <= inA - inB; // code 3 does subtraction
      4'b0100 : out <= inA ^ inB; // code 4 does a bitwise xor
    endcase
  end

  assign zero = (out==0) ? 1:0;   // if out == 0, then raise zero flag

endmodule
`endif
```

Figure 7: Verilog implementation of ALU with its testbench

In the test bench, we carry out the bitwise AND and the bitwise OR operations with A = 0x0000FFFF and B = 0xFFFF0000. As expected, the output wave shows 0xFFFFFFFF and

0x00000000 for the respective operations. Notice, the zero flag is enabled for the second result. The remaining inputs demonstrate the functionality for the remaining operations (addition, subtraction, and bitwise XOR).

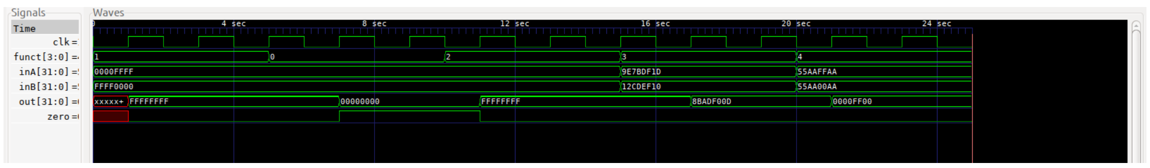
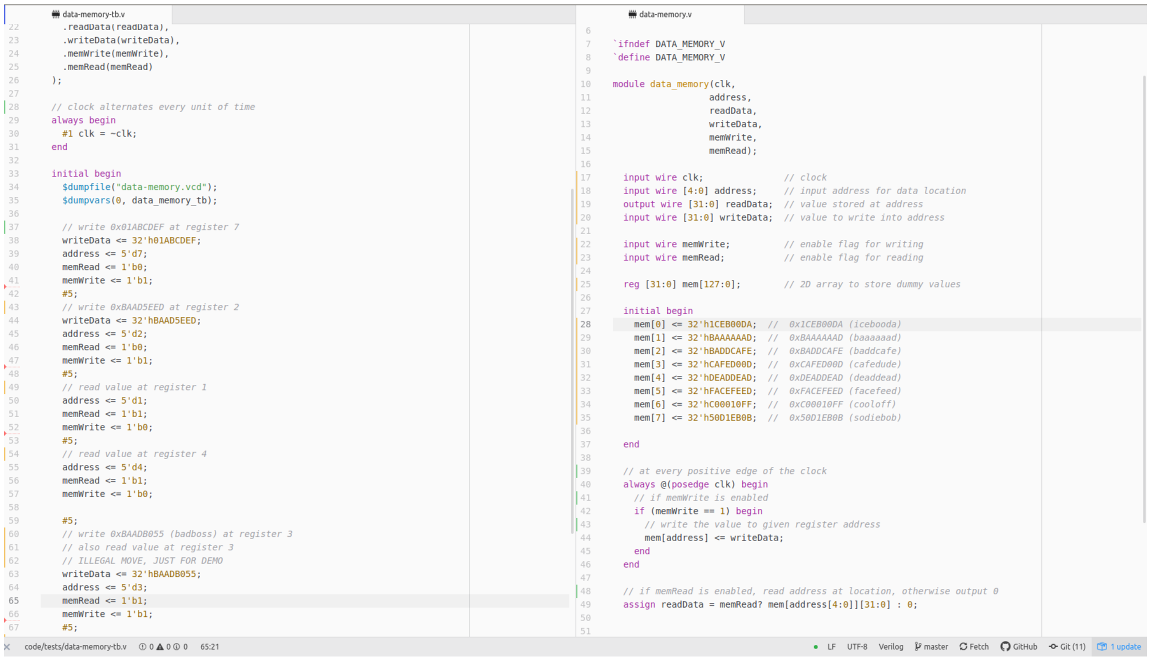


Figure 8: Waveform of the ALU module

3.5 Data Memory

Driven by a clock, the data memory receives an address to which it either reads or writes. Hence, there are two control signals memRead and memWrite, which decide the operation of the data memory. Given that memWrite is enabled, the data memory also accepts a 32-bit value to write to the given address.

This data memory implementation simulates memory by using registers, which are instantiated with some random values. At the positive edge of each clock cycle, if memWrite is enabled, the module writes to the given address. It then checks for the memRead signal and outputs 0 if it is disabled.



data-memory-tb.v

```verilog
    .readData(readData),
    .writeData(writeData),
    .memWrite(memWrite),
    .memRead(memRead)
  );

  // clock alternates every unit of time
  always begin
    #1 clk = ~clk;
  end

  initial begin
    $dumpfile("data-memory.vcd");
    $dumpvars(0, data_memory_tb);

    // write 0x01ABCDEF at register 7
    writeData <= 32'h01ABCDEF;
    address <= 5'd7;
    memRead <= 1'b0;
    memWrite <= 1'b1;
    #5;
    // write 0xBAAD5EED at register 2
    writeData <= 32'hBAAD5EED;
    address <= 5'd2;
    memRead <= 1'b0;
    memWrite <= 1'b1;
    #5;
    // read value at register 1
    address <= 5'd1;
    memRead <= 1'b1;
    memWrite <= 1'b0;
    #5;
    // read value at register 4
    address <= 5'd4;
    memRead <= 1'b1;
    memWrite <= 1'b0;

    #5;
    // write 0xBAADB055 (badboss) at register 3
    // also read value at register 3
    // ILLEGAL MOVE, JUST FOR DEMO
    writeData <= 32'hBAADB055;
    address <= 5'd3;
    memRead <= 1'b1;
    memWrite <= 1'b1;
    #5;
```

code/tests/data-memory-tb.v 0 0 0 65:21

data-memory.v

```verilog

`ifndef DATA_MEMORY_V
`define DATA_MEMORY_V

module data_memory(clk,
                   address,
                   readData,
                   writeData,
                   memWrite,
                   memRead);

  input wire clk;               // clock
  input wire [4:0] address;     // input address for data location
  output wire [31:0] readData;  // value stored at address
  input wire [31:0] writeData;  // value to write into address

  input wire memWrite;          // enable flag for writing
  input wire memRead;           // enable flag for reading

  reg [31:0] mem[127:0];        // 2D array to store dummy values

  initial begin
    mem[0] <= 32'h1CEB00DA;  //  0x1CEB00DA (icebooda)
    mem[1] <= 32'hBAAAAAAD;  //  0xBAAAAAAD (baaaaaad)
    mem[2] <= 32'hBADDCAFE;  //  0xBADDCAFE (baddcafe)
    mem[3] <= 32'hCAFED00D;  //  0xCAFED00D (cafedude)
    mem[4] <= 32'hDEADDEAD;  //  0xDEADDEAD (deaddead)
    mem[5] <= 32'hFACEFEED;  //  0xFACEFEED (facefeed)
    mem[6] <= 32'hC00010FF;  //  0xC00010FF (cooloff)
    mem[7] <= 32'h50D1EB0B;  //  0x50D1EB0B (sodiebob)

  end

  // at every positive edge of the clock
  always @(posedge clk) begin
    // if memWrite is enabled
    if (memWrite == 1) begin
      // write the value to given register address
      mem[address] <= writeData;
    end
  end

  // if memRead is enabled, read address at location, otherwise output 0
  assign readData = memRead? mem[address[4:0]][31:0] : 0;


```

LF UTF-8 Verilog master Fetch GitHub Git (11) 1 update

Figure 9: Verilog implementation of data memory with its testbench

Looking at the testbench, we first write a value to the 7th registe. We then write another value to the 2nd register; both times the memWrite signal is enabled, so the write is successful. Notice that the readData signal outputs 0 because the memRead control is disabled. In the next two assignments, we read from registers 1 and 4.

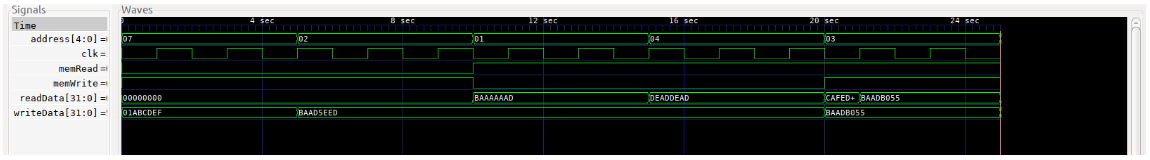


Figure 10: Waveform of the data memory module

Finally, we attempt to read and write in the same cycle, which is an illegal move. The output waves demonstrates this issue, because the readData changes, giving two values within the same instruction cycle. This could adversely affect the CPU operation; hence, only one of the two control signals can be enabled at a given time.

4 Conclusion

The above implementations generate the basic modules of the MIPS CPU. Future work involves connecting these modules as well as covering corner cases to allow robust functionality